Full-edged Real-Time Indexing for Constant Size Alphabets

Gregory Kucherov  
CNRS/LIGM Marne-la-Vallée, France

Yakov Nekrich  
University of Kansas, USA

June 23, 2013
History of the Problem and Related Work
find all occurrences of a pattern $P$ in a text $T$

- string matching: $P$ is fixed (or given first)
- indexing: $T$ is fixed (or given first)
Real-time string matching: early results

- For a fixed $P$, string matching can be done in real time independently on the alphabet size [Knuth-Morris-Pratt 77] (see also [Matiyasevich 71 (in russian)])

- Language $\{P \# T : P \text{ occurs in } T\}$ can be recognized in real time by a Turing machine [Galil, JACM 81]
Real-time string matching: early results

- for a fixed $P$, string matching can be done in real time independently on the alphabet size [Knuth-Morris-Pratt 77] (see also [Matiyasevich 71 (in russian)])
- language $\{P\#T : P \text{ occurs in } T\}$ can be recognized in real time by a Turing machine [Galil, JACM 81]
- language $\{T\#P : P \text{ occurs in } T\}$ cannot be recognized in real time by (multi-tape) TM [Freidzon 68 (in russian)]
- ... but this can be done by a TM with two-dimensional tape if $T$ and $P$ are submitted on two independent input tapes [Matiyasevich 71]
Indexing under RAM model

- \{ T\#P : P occurs in T \} can be recognized in real time on RAM [Slisenko 76-78]
- same result in [Kosaraju STOC 94]
- there is an index of T that can be updated in real time such that for any pattern query P made at any moment, one can check of P occurs in current T in time \( O(|P|) \) [Amir, Nor SODA 08]. The result assumes a constant-size alphabet.
Indexing under RAM model

- \( \{ T \neq P : P \text{ occurs in } T \} \) can be recognized in real time on RAM [Slisenko 76-78]
- same result in [Kosaraju STOC 94]
- there is an index of \( T \) that can be updated in real time such that for any pattern query \( P \) made at any moment, one can check if \( P \) occurs in current \( T \) in time \( O(|P|) \) [Amir,Nor SODA 08]. The result assumes a constant-size alphabet.

**Our result**: an index that can be updated in real time and all occurrences of \( P \) in the current text are reported in time \( O(|P| + nb\_occ) \). The result assumes a constant-size alphabet.
Suffix Tree

Context and history

Supporting real-time
Suffix Tree

Three classical linear-time algorithms for constructing a suffix tree

- [Weiner 73]: right-to-left construction
- [McCreight 76]: left-to-right
- [Ukkonen 95]: left-to-right online
Suffix Tree

Three classical linear-time algorithms for constructing a suffix tree

- [Weiner 73]: right-to-left construction
- [McCreight 76]: left-to-right
- [Ukkonen 95]: left-to-right online

Weiner is more suitable for real-time as only a constant number of changes is made at each letter
Suffix Tree

Three classical linear-time algorithms for constructing a suffix tree

- [Weiner 73]: right-to-left construction
- [McCreight 76]: left-to-right
- [Ukkonen 95]: left-to-right online

Weiner is more suitable for real-time as only a constant number of changes is made at each letter.

McCreight and Ukkonen use *suffix links*:

for every node $v$, $v = au$, define suffix link $S(v) = u$

Weiner uses *prefix links*:

for every node $v$, and for every letter $a$, $P_a(v) = av$

provided that node $av$ exists
Prefix links

- The target of a prefix link can be an explicit or an implicit node. The prefix link is called respectively hard or soft.
- If a node has a prefix link by letter $a$, then all its ancestors do too.
- A soft link $P_a(v)$ is defined iff there is a unique closest descendant $u$ such that $P_a(u)$ is hard, and $P_a(v)$ points to edge $(w, P_a(u))$. 

![Diagram showing prefix links in a suffix tree](image_url)
Main idea of Weiner algorithm

transforming suffix tree for $t$ to suffix tree for $at$

- find the lowest ancestor $u$ of $t$ with a prefix link $P_a(u)$
- $P_a(u)$ is the branching point

$abbabac \Rightarrow babbabac$
Main idea of Weiner algorithm

transforming suffix tree for $t$ to suffix tree for $at$

- find the lowest ancestor $u$ of $t$ with a prefix link $P_a(u)$
- $P_a(u)$ is the branching point

$abbabac \Rightarrow babbabac$
Towards real-time construction of suffix tree

- [Amir, Kopelowitz, Lewenstein, Lewenstein SPIRE 05]: $O(\log n)$ worst-case per symbol, unbounded alphabet
- [Breslauer, Italiano SPIRE 11]: $O(\log \log n)$ worst-case per symbol, constant alphabet
- [Kopelowitz FOCS 12]: $O(\log \log n + \log \log \sigma)$ expected worst-case per symbol, unbounded alphabet
- [Fischer, Gawrychowski arxiv 13]: $O(\log \log n + \frac{\log^2 \log \sigma}{\log \log \log \sigma})$ worst-case per symbol, unbounded alphabet
Our implementation of Weiner algorithm
Main ideas:

- we store only hard prefix links, soft links are computed “on the fly”
- we maintain a list $\mathcal{L}_W$ corresponding to the Euler tour of the tree
- each node with defined hard link $\mathcal{W}_a(u)$ is “colored” by $a$ in $\mathcal{L}_W$
Main ideas:

- we store only hard prefix links, soft links are computed “on the fly”
- we maintain a list \( \mathcal{L}_W \) corresponding to the Euler tour of the tree
- each node with defined hard link \( \mathcal{W}_a(u) \) is “colored” by \( a \) in \( \mathcal{L}_W \)

Lemma: To find the deepest ancestor \( u \) of \( t \) with defined (possibly soft) link \( \mathcal{W}_a(u) \), let \( v_1 \) (resp. \( v_2 \)) be the closest node colored with \( a \) preceding (resp. following) \( t \) in \( \mathcal{L}_W \). Then \( u \) is the deepest node between \( lca(t, v_1) \) and \( lca(t, v_2) \).
Our implementation of Weiner

Main ideas:

- we store only hard prefix links, soft links are computed “on the fly”
- we maintain a list $L_W$ corresponding to the Euler tour of the tree
- each node with defined hard link $\mathcal{W}_a(u)$ is “colored” by $a$ in $L_W$

Lemma: To find the deepest ancestor $u$ of $t$ with defined (possibly soft) link $\mathcal{W}_a(u)$, let $v_1$ (resp. $v_2$) be the closest node colored with $a$ preceding (resp. following) $t$ in $L_W$. Then $u$ is the deepest node between $lca(t, v_1)$ and $lca(t, v_2)$.
Main ideas:

- we store only hard prefix links, soft links are computed “on the fly”
- we maintain a list $L_W$ corresponding to the Euler tour of the tree
- each node with defined hard link $W_a(u)$ is “colored” by $a$ in $L_W$

Lemma: To find the deepest ancestor $u$ of $t$ with defined (possibly soft) link $W_a(u)$, let $v_1$ (resp. $v_2$) be the closest node colored with $a$ preceding (resp. following) $t$ in $L_W$. Then $u$ is the deepest node between $lca(t, v_1)$ and $lca(t, v_2)$.
Tools that we will use

Colored Predecessor in a List

**Problem**: Maintain a dynamic list $\mathcal{L}$ whose elements are assigned “colors”. Support queries: given an element $e \in \mathcal{L}$ and a color $col$, retrieve the closest element $e' \in \mathcal{L}$ preceding $e$ with color $col$.

**Theorem** [Mortensen SODA 03, Giyora, Kaplan 09]: If the number of colors is smaller than $\log^{1/4} n$, then there exists a $O(|\mathcal{L}|)$ data structure that answers colored predecessor queries in $O(\log \log |\mathcal{L}|)$ time and supports updates (insertions) in $O(\log \log |\mathcal{L}|)$ time.
Dynamic Lowest Common Ancestor (LCA)

**Problem**: Maintain a dynamic tree (leave insertion/deletion, edge split, edge merge) supporting lowest common ancestor of two nodes

**Theorem** [Cole, Hariharan 05]: both updates and queries can be supported in worst-case $O(1)$ time
What we obtained so far

**Theorem**

We can maintain a suffix tree of right-to-left streaming text by spending $O(\log \log n)$ worst-case time on each symbol, assuming an alphabet size $\leq \log^{1/4} n$.

Simplifies and (slightly) generalizes [Breslauer, Italiano 11]
Our solution to real-time text indexing
Our implementation of ST

Supporting real-time

Fully real-time text indexing on constant-size alphabet

Main idea:

Maintain three distinct data structures for patterns of length

- $\geq \log^2 \log n$ (long patterns),
- between $\log^2 \log \log n$ and $\log^2 \log n$ (medium-size patterns),
- $\leq \log^2 \log \log n$ (small patterns)
Group text symbols into meta-symbols of size $d = \log \log n/(4 \log \sigma)$. There are $\sigma^d = \log^{1/4} n$ meta-symbols.
Data structure for long patterns (sketch)

- Group text symbols into meta-symbols of size 
  \[ d = \log \log n / (4 \log \sigma) \]. There are \( \sigma^d = \log^{1/4} n \) meta-symbols.

- Apply the suffix tree construction. Spend \( O(\log \log n) \) time on each meta-symbol (i.e. amortized \( O(1) \) time on each symbol).
Data structure for long patterns (sketch)

- Group text symbols into meta-symbols of size $d = \log \log n / (4 \log \sigma)$. There are $\sigma^d = \log^{1/4} n$ meta-symbols.
- Apply the suffix tree construction. Spend $O(\log \log n)$ time on each meta-symbol (i.e. amortized $O(1)$ time on each symbol).
- To match a long pattern $P$, consider all offsets $\delta$, $0 \leq \delta \leq d - 1$. For each $\delta$, $P$ can be matched in time $|P|/d + \log \log n$ (details left out).
Data structure for long patterns (sketch)

- Group text symbols into meta-symbols of size $d = \log \log n/(4 \log \sigma)$. There are $\sigma^d = \log^{1/4} n$ meta-symbols.
- Apply the suffix tree construction. Spend $O(\log \log n)$ time on each meta-symbol (i.e. amortized $O(1)$ time on each symbol).
- To match a long pattern $P$, consider all offsets $\delta$, $0 \leq \delta \leq d - 1$. For each $\delta$, $P$ can be matched in time $|P|/d + \log \log n$ (details left out).
- Overall we obtain time
  $O(d(|P|/d + \log \log n) + nb\_occ) = O(|P| + nb\_occ)$ as $|P| \geq \log^2 \log n$
Group text symbols into meta-symbols of size $d = \log \log \log n$. Maintain compacted trie of truncated suffixes of length $\log^2 \log n$ considered over the alphabet of meta-symbols.

Number of suffixes (trie leaves) is $O(d^{\log^2 \log n})$.

To maintain this trie, use (basically) the same algorithm as above. Spend $O(\log \log (d^{\log^2 \log n})) = O(\log \log \log n)$ time on each truncated suffix (i.e. amortized $O(1)$ time on each letter).

Matching a medium-size pattern $P$ is done similarly. The overall time is

$O(d(|P|/d + \log \log \log n) + nb_{occ}) = O(|P| + nb_{occ})$ as $|P| \geq \log^2 \log \log n$. 
Data structure for small patterns (idea)

- Maintain a tree of truncated suffixes of length $\log^2 \log \log n$ and a list of occurrences of each truncated suffix in the current text
- Tabulate all possible trees and all possible updates
- Every update takes $O(1)$ time and a matching query takes time $O(|P| + nb\_occ)$
Turning it fully real-time

Two more problems should be overcome to make this solution real-time

Problem 1: Most recent blocks should have a special treatment.
Turning it fully real-time

Two more problems should be overcome to make this solution real-time

Problem 1: Most recent blocks should have a special treatment.
Problem 2: Text length $n$ is unknown.
For a streaming text over a constant-size alphabet, there exists a data structure that can be updated in real time such that at any moment, all positions of any pattern $P$ in the current text can be reported in time $O(|P| + nb_{occ})$. 